Flexible Dynamic Linking

Alex Buckley and Sophia Drossopoulou
Department of Computing, Imperial College London
{abuckley,sd}@doc.ic.ac.uk

Abstract

Dynamic linking, as in Java and C#, allows users to execute the most recent versions of software without re-compilation or re-linking. Dynamic linking is guided by type names stored in the bytecode.

In current dynamic linking schemes, these type names are hard-coded into the bytecode. Thus, the bytecode reflects the compilation environment that produced it. However, the compilation environment need not be the same as the execution environment: a class may be replaced by one that offers the "same" services but has a different name. Such changes are not supported by current linking schemes.

We suggest a more flexible approach to dynamic linking, where bytecode contains type variables rather than types, and where these type variables are substituted during execution. We develop a non-deterministic system that allows type variable substitution at many different points, and sketch a proof of soundness.

1 Introduction

Java and C# support a dynamic linking process, allowing programs to take advantage of newly-updated classes and reducing startup times. The process is well-documented for Java [LY99]: a `.class` file is loaded from disk or the Internet, verified according to structural and semantic rules, and prepared in memory; eventually the class's references to other classes' fields and methods are resolved.

To support verification and resolution, bytecode for field access and method call includes *type annotations*. A compiler generates these annotations using type information from available classes. However, the compilation environment may be different to the execution environment in which the bytecode runs. By 'hard-coding' class names into bytecode, a compiler makes eager choices about which classes the JVM will link at run-time. If the classes available at run-time differ from those available at compile-time, a program may experience run-time errors purely because of overly restrictive signatures in its bytecode.

In contrast, we aim to perform dynamic linking as flexibly and lazily as possible:

- **To provide flexibility**, we allow bytecode's type annotations to mention *type variables* rather than actual class names. The linker substitutes class names for type variables lazily at run-time.
- **To increase laziness**, we allow the verifier - which may have to check subtype relationships involving type variables - to assume that any necessary subtyping holds, until such time that actual class loading contradicts the assumptions (if at all). This follows the scheme in [QGC00].

For the purpose of this work, we are not concerned with the compilation process that places annotations with type variables into bytecode. Nor are we concerned with how the run-time linker chooses classes to substitute for type variables. Our concern is how late the type variables can be substituted and what assumptions need to be checked when, while still guaranteeing sound execution. Sound execution preserves program well-formedness and types (under substitution) of well-formed expressions.

2 Flexible dynamic linking with type variables

In this section, we review the use of type variables in separate compilation (as suggested by [ADDZ04]) and identify their role in a dynamic linking process more flexible than that of [LY99].

Throughout, we use the following notation for bytecode instructions with type annotations: $\texttt{.f}(\!|\texttt{T}_d, \texttt{T}_f|\!)$ for an access to field `f` of static type $\texttt{T}_f$ visible in type $\texttt{T}_d$; $\texttt{.m}(\!|\texttt{T}_d, \texttt{T}_r, \texttt{T}_p|\!)$ for an invocation of method `m` visible in type $\texttt{T}_d$ with static return type $\texttt{T}_r$ and static formal parameter type $\texttt{T}_p$; $\texttt{new C}(\bar{\texttt{e}})(\!|\bar{\texttt{T}}|\!)$ for an instantiation of an object of class `C` with parameters $\bar{\texttt{e}}$ of static types $\bar{\texttt{T}}$.

2.1 Type variables for flexible separate compilation

Current Java compilers require that all classes used by a program are present during compilation. They have no standard mechanism for discovering or obtaining more classes, in sharp contrast to a JVM which can dynamically download classes on demand.

Consider a program that can use optional plugins from third parties, which cannot be redistributed with the program. Today, the programmer must not mention any class or interface names from the plugins in his code, because while he will be able to compile the program, users without the plugins will not. It is straightforward to avoid class identifiers since the `Class.forName()` method can instantiate a class given its name as a `String`. But to avoid interface names, the programmer must manipulate all of a plugin's objects as if they were of class `Object` and invoke methods through reflection, thus losing static typechecking.

In our scheme, type variables can support separate compilation by allowing bytecode to be less tightly bound to other classes. A compiler that cannot find a referenced class could emit bytecode featuring type variables in type annotations and rely on a dynamic linker to substitute them to class names.[1] This may enable a programmer to do more development and testing than if he has to wait for classes mentioned in his code to become available.

If in fact all classes needed by a compiler are present, there is a tight dependency between those classes and emitted bytecode. Suppose a method invocation is compiled to $\texttt{.m}(\!|\texttt{C}, \texttt{Object}, \texttt{String}|\!)$. This embeds the structure of the class `C` available to the compiler into the bytecode, namely the requirement that `C` have a method `m` taking a `String` and returning an `Object`. If the `C` available at run-time has a method `m` with a narrower return type or wider argument type, method resolution will fail with a `NoSuchMethodError` exception.

A compiler supporting type variables could emit $\texttt{.m}(\!|\texttt{C}, \texttt{X}, \texttt{Y}|\!)$ and rely on a dynamic linker to find a `C` whose method `m` provides appropriate return and argument types to substitute for `X` and `Y`.

2.2 Type variables for flexible dynamic linking

An execution environment may provide more classes than those available at compilation, *e.g.*

- As well as the standard `ArrayList` and `LinkedList` classes, the user may have other implementations of the `List` interface from the Java Collections Framework [JCF02] in their `CLASSPATH`.
- A database server may have a vendor-provided implementation of a generic database access interface, offering client programs a faster and more secure connection to the database.
- A Web server may supply custom compression libraries to servlets as well as `java.util.zip`.

Generally, a program cannot benefit from extra classes being available at run-time unless it uses rather complex and slow reflection schemes. The *Factory* pattern is no solution since a *Factory* will hard-code interface or class identifiers even if its clients do not. Our scheme enables the use of extra classes because, as noted above, type annotations in bytecode can refer not to actual classes but rather to type variables that undergo substitution at run-time. We therefore suggest that type variables and a suitable dynamic linker provide flexibility that would otherwise have to be programmed explicitly.

[1] We expect the compiler would also have to emit type assumptions [ADDZ04] for the linker to use.

2.3 The mechanics of flexible dynamic linking

The listings below show two execution environments, each consisting of bytecode for five classes.[2] Listing 1 features ordinary class names while listing 2 features type variables `X`, `Y`, `Z`. Fig. 1 shows dynamic linking in each execution environment. We list the linking steps involved with each line, and denote with ↪ any necessary sub-steps, *e.g.* in order to execute `new A`, we have to load and verify class `A`.

```
1  class Test {
2   ...
3   new A()[].f[A,B].m[B,C,D](new E()[]);
4   ...
5  }
6
7  class A { B f; }
8  class B { C m(D d) {} }
9
10 class D {}
11 class E extends D {}
```

Listing 1: Without type variables

```
1  class Test {
2   ...
3   new A()[].f[A,X].m[X,Y,Z](new M()[]);
4   ...
5  }
6
7  class A { J f; }
8  class J { K m(L l) {} }
9
10 class L {}
11 class M extends L {}
```

Listing 2: With type variables

Non-flexible linking steps	**Flexible linking steps**
Load `Test`	Load `Test`
Verify `Test`	Verify `Test`
↪Check `E` $\leq$ `D`	↪Assume `M` $\leq$ `Z`
↪Load `E`	
↪Throw `VerifyError` if `E` $\not\leq$ `D`	
Execute `new A`	Execute `new A`
↪Load `A`, Verify `A`	↪Load `A`, Verify `A`
Resolve field `f` in `A`	Resolve field `f` in `A`
↪Throw `NoSuchFieldError` if `f` is not a `B`	↪Substitute `X` to `J`
↪Load `B`, Verify `B`	↪Load `J`, Verify `J`
	Substitute `Y` to `K` and `Z` to `L`
Execute `B::m` method call	Execute `J::m` method call
↪Execute `new E`	↪Execute `new M`
↪Verify `E`	↪Load `M`
↪Load `D`, Verify `D`	↪Check assumption `M` $\leq$ `L`
	↪Load `L`, Verify `L`
	↪Verify `M`
↪Call `m`	↪Call `m`

Figure 1: Non-flexible linking for Listing 1 and Flexible linking for Listing 2

The main differences between flexible and non-flexible linking steps are:

- Flexible linking can make assumptions about subtypes during verification, whereas non-flexible linking must load classes to check subtypes.

  During verification of class `Test`, non-flexible linking has to immediately load class `E` to check whether it subtypes `D`, the formal parameter type expected by the method call. Flexible linking postpones class loading by assuming that the type of the actual parameter, `M`, is a subtype of `Z`, the formal parameter type mentioned in the annotation. This assumption does not need to be checked immediately nor does it require `Z` to be substituted to a class.

[2] We show bytecode at a high level, resembling source code, as in [DLE03].

`Z` must be substituted by the time `m` is called (in order to use the parameter type from the bytecode in method selection), or by the time `M` is loaded (in order to allow the linker to check the assumption that `M` subtypes `Z`).

- Flexible linking allows code in one class to be less tightly bound to the definition of another class.

  With non-flexible linking, resolution must find a field `B f` in class `A` or an exception is thrown. Flexible linking allows bytecode in `Test` to access `f` at a type variable `X`; the flexible linker substitutes `X` to `J`.

- Type variables in bytecode allow the linker to link the "best" class available at run-time, whereas non-flexible linking must follow the bytecode's requirements for classes exactly.

3 The Model

3.1 Program definitions

We model dynamic linking as the evolution of a program. When the linker loads a class, the program stores the class's name and definition read from the `.class` file. When the verifier verifies a class,[3] the program keeps track of verified method bodies. Assumptions about subtypes made in support of verification are added to the program as necessary, and checked when classes they concern are loaded. Consequently, a program is a triple holding subtype assumptions in $\mathbb{P}_{\mathbb{A}}$, the verified methods of a class in $\mathbb{P}_{\mathbb{B}}$, and class definitions in $\mathbb{P}_{\mathbb{C}}$. The syntax of programs and expressions is given in fig. 2 in appendix A.[4]

We use a simple language syntax, based on Featherweight Java [IPW99], in order to concentrate on linking rather than language features. The metavariables `A`,`B`,`C` range over class names, `X` and `Y` over type variables, `m` over method names, `f` over field names, and `e` over expressions. `T` ranges over types, which are class names or type variables. $\mathbb{P}_{\mathbb{C}}$ maps class names to class declarations.[5] Unlike FJ, where the table of class declarations is fixed, $\mathbb{P}_{\mathbb{C}}$ grows via program extension as classes are loaded. We also drop constructors from class declarations and assume that an object's fields are initialised implicitly at instantiation. We support field hiding and method overloading and overriding. $\mathbb{P}_{\mathbb{B}}$ maps classes to verified method signatures and bodies.[6] $\mathbb{P}_{\mathbb{A}}$ holds assumptions of the form $(\mathtt{T}, \mathtt{T}')$, meaning "`T` is assumed to be a subtype of `T′`".[7] Like $\mathbb{P}_{\mathbb{C}}$, both $\mathbb{P}_{\mathbb{B}}$ and $\mathbb{P}_{\mathbb{A}}$ grow via program extension.

We use FJ-style syntax to represent bytecode expressions, similar to [AZ04]. Field access, method call and object instantiation expressions are as described at the start of section 2.[8] We drop casting and, unlike previous work [DLE03], do not model offsets or the resolution stage of dynamic linking; only class loading, verification and expression evaluation stages are relevant here.

3.2 Subtyping

Subtyping is shown in fig. 3 and has two forms:

Definite subtypes are the reflexive and transitive closure of classes in $\mathbb{P}_{\mathbb{C}}$ under ordinary inheritance. The judgement $\mathbb{P} \vdash \mathtt{C} \leq \mathtt{C}'$ judges that `C` is a *definite subtype* of `C′`.

[3] Verification is simplified to type-checking and does not check `.class` file integrity as in [LY99].

[4] We implicitly switch between sequences and sets of tuples.

[5] We expect that $(\mathtt{C}, \mathtt{CL}), (\mathtt{C}, \mathtt{CL}') \in \mathbb{P}_{\mathbb{C}} \implies \mathtt{CL} = \mathtt{CL}'$, and thus $\mathbb{P}_{\mathbb{C}}(\mathtt{C})$ has the obvious meaning.

[6] The notation $\mathbb{P}_{\mathbb{B}}(\mathtt{C})$ gives all the tuples (`m`, `T`, `T′`, `e`) such that (`C`, `m`, `T`, `T′`, `e`) $\in\mathbb{P}_{\mathbb{B}}$. Furthermore we require that (`m`, `T`, `T′`, `e`) $\in\mathbb{P}_{\mathbb{B}}(\mathtt{C})$ and (`m`, `T`, `T′`, `e′`) $\in\mathbb{P}_{\mathbb{B}}(\mathtt{C}) \implies \mathtt{e}{=}\mathtt{e}'$ and thus $\mathbb{P}_{\mathbb{B}}(\mathtt{C})(\mathtt{m}, \mathtt{T}, \mathtt{T}')$ has the obvious meaning.

[7] We write (`T`,`T′`) $\in\mathbb{P}_{\mathbb{A}}$ iff $\mathbb{P}_{\mathbb{A}}$ = ..., (`T`,`T′`), ...

[8] In method call, we require methods to take a single parameter `y`, unlike FJ.

Assumptive subtypes are the transitive closure of assumptions in $\mathbb{P}_{\mathbb{A}}$ together with all definite subtypes.[9] The judgement $\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}'$ judges that $\mathtt{T}$ is an *assumptive subtype* of $\mathtt{T}'$.

A class may be both a definite subtype and an assumptive subtype (respectively, supertype), but a type variable may only be an assumptive subtype or supertype, *i.e.* neither $\mathbb{P} \vdash \mathtt{X} \leq \mathtt{C}$ nor $\mathbb{P} \vdash \mathtt{C} \leq \mathtt{X}$ can be deduced.

3.3 Typing and Verification

The typing judgements are shown in fig. 4. They have the form $\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T}$ and are used to verify a class during program extension. They use assumptive subtypes because of the need to verify expressions featuring type variables; appropriate assumptions may have to be added before a typing judgement is successful. On the other hand, if definite subtypes are known, then the typing rules can use them automatically by Subtyping Rule 5.

The type of a formal parameter `y` is looked up in the type environment Γ. The type of an instantiation expression is the type being instantiated, and the initial field values $\bar{\mathtt{e}}$ must be assumptive subtypes of the annotated types $\bar{\mathtt{T}}$. For field access, the receiver must be well-typed and an assumptive subtype of the defining type mentioned in the bytecode. For method call, the receiver and argument must be well-typed and be assumptive subtypes of the defining type and parameter type mentioned in the bytecode.

3.4 Program extension

A program $\mathbb{P}'$ extends a program $\mathbb{P}$ if $\mathbb{P}'$ contains more information (through loading of classes or addition of assumptions) or more refined information (through verification of classes) than $\mathbb{P}$. Program extension is shown in fig. 5 and has the form $\vdash \mathbb{P}' \leq \mathbb{P}$. A program can be extended at any time with any of the following linking steps:

Class loading Rule 1 describes extension via class loading. It adds a single class declaration $\mathtt{CL_C}$ to $\mathbb{P}_{\mathbb{C}}$. The loaded class' superclass must have already been loaded. Assumptions are checked: if the non-extended program $\mathbb{P}$ judged that the class `C` being loaded was an assumptive subtype of some type `T`, then `C` must be a definite subtype of `T` in the program with `C` loaded, *i.e.* in $\mathbb{P} \oplus \mathtt{CL_C}$.

Class verification Rule 2 describes extension via verification of a class, by verifying all its methods at the same time. Verification uses the typing rules from fig. 4 to type a method's body in the context of a type environment Γ that maps `this` to the class being verified and the formal parameter `y` to the declared formal parameter type $\mathtt{T}_p$. As noted above, assumptions may need to be added to ensure verification does not get stuck.

The fact that a class `C` has been verified successfully is recorded by storing it in $\mathbb{P}_{\mathbb{B}}$ with its superclass's and its own verified methods, *i.e.* $\mathbb{P} \oplus \mathtt{CV_C}$. Overriding is supported by the extension operator $\bullet$ that prefers to store a method from a newly verified class rather than its superclass:

$$M \bullet (\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p \mapsto \mathtt{e}) = \begin{cases} (M \backslash (\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p \mapsto \_)), (\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p \mapsto \mathtt{e}) & if\ (\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p) \in dom(M) \\ M, (\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p \mapsto \mathtt{e}) & otherwise \end{cases}$$

Assumption addition Rules 3 and 4 describe extension via the addition of an assumption to $\mathbb{P}_{\mathbb{A}}$, *i.e.* $\mathbb{P} \oplus (\mathtt{T}, \mathtt{T}')$. For a the class `C` to be an assumptive subtype of some type `T`, *i.e.* (`C`, `T`), `T` must not itself be an assumptive subtype of `C`. This conservative rule prevents contradiction of known subtyping and the formation of an illegal class hierarchy. For example, if `B` is already a definite subtype of `A`, then the assumption (`A`, `B`) is not permitted. We always allow an assumption where a type variable is the subtype.

[9] Note that we need transitivity rules for both definite and assumptive subtypes, but only one reflexive rule. This is because *any* type is a definite subtype of itself, so $\mathbb{P} \vdash \mathtt{X} \leq \mathtt{X}$ always holds and rule 5 automatically gives $\mathbb{P} \vdash_{\mathbb{A}} \mathtt{X} \leq \mathtt{X}$.

3.5 Execution

Execution has the form $\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}', \mathtt{e}'$ and is shown in fig. 6. Field access looks up the field in the class $\mathtt{C}_d$ mentioned in the bytecode's type annotation, rather than the dynamic type $\mathtt{C}$ of the object. Method call looks up the method in the dynamic type $\mathtt{C}$ of the receiver.[10]

Rules 3 and 4 model flexible dynamic linking. Rule 3 allows program extension at any point during execution. Rule 4 allows a type variable to be substituted by a class name if the substitution is well-formed. The substitution is applied to the expression and globally throughout the program: to the assumptions in $\mathbb{P}_{\mathbb{A}}$ [11], to the classes, fields and verified methods in $\mathbb{P}_{\mathbb{B}}$, and to the class declarations in $\mathbb{P}_{\mathbb{C}}$.

Well-formedness of substitutions is defined in fig. 7 and has the form $\mathbb{P} \vdash \sigma \diamond$. A substitution from type variable $\mathtt{X}$ to class name $\mathtt{C}$, written $\mathtt{C}/\mathtt{X}$, is well-formed if $\mathtt{C}$ is in the same position in the class hierarchy as $\mathtt{X}$ was assumed to be. The identity, *i.e.* $\mathtt{T}/\mathtt{T}$, is also a well-formed substitution.

4 Soundness

Fig. 8 expresses the requirements for well-formed programs: 1) the `Object` class does not feature in the table of class declarations, 2) all superclasses are loaded, 3) the class hierarchy is well-formed, *i.e.* definite subtypes are not circular, and 4) all loaded classes that have been verified have well-typed methods, each returning an assumptive subtype of the expected return type.[12]

Appendix B lists the properties that we expect our system to satisfy, though we do not have proofs yet. Conjecture 1 states that execution causes program extension modulo substitution. Conjecture 2 states that properties of definite subtyping, typing and programs well-formedness are preserved modulo well-formed substitutions. Conjecture 3 states that program extension preserves types, subtypes and program well-formedness. Therefore, execution preserves program well-formedness.

Conjecture 4 states that execution does not affect the type of unevaluated expressions; execution may perform substitutions such that unevaluated expressions originally typed as a type variable now type as a class after execution, but there exists a well-formed substitution from the type variable to the class.

Subject reduction states that an evaluated expression preserves its type, modulo well-formed substitutions, after an execution step:

Conjecture 5 Subject reduction.

If $\vdash \mathbb{P}$ and $\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T}$ and $\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}', \mathtt{e}'$

then

$\vdash \mathbb{P}'$ and

$\exists \mathtt{T}', \sigma\colon \mathbb{P} \vdash \sigma \diamond$ and $\mathbb{P}', \Gamma \vdash \mathtt{e}' : \mathtt{T}'$ and $\mathbb{P}' \vdash \mathtt{T}' \leq \mathtt{T}[\sigma]$

5 Discussion

The work presented in this paper is the outcome of many iterations. The main issues which caused the iterations were an investigation of the possible role of type variables and the design of the formal system.

[10] Note that if the verifier is fooled or disabled, field access expressions can return values that are not of the expected type. For example, the expression $\mathtt{new\ C(e)}\langle\!\langle\mathtt{A}\rangle\!\rangle\ \mathtt{.f}\langle\!\langle \mathtt{C}_d, \mathtt{C}_f\rangle\!\rangle$ has expected type $\mathtt{C}_f$. But without correct verification, if $\mathtt{C}_d$ has a single field $\mathtt{f}$ of type $\mathtt{C}_f$ and $\mathtt{C}$ has a single field $\mathtt{f}$ of type $\mathtt{A}$, it will evaluate to $\mathtt{e}$ which is of type $\mathtt{A}$ - even if $\mathbb{P} \not\vdash \mathtt{C} \leq \mathtt{C}_d$ and $\mathbb{P} \not\vdash \mathtt{A} \leq \mathtt{C}_f$.

[11] Applying substitution $[\mathtt{C}/\mathtt{X}]$ to assumption $(\mathtt{T}, \mathtt{T}')$ gives assumption $(\mathtt{T}[\mathtt{C}/\mathtt{X}], \mathtt{T}'[\mathtt{C}/\mathtt{X}])$.

[12] Recall that any substitutions made during execution will have caused $\mathbb{P}_{\mathbb{B}}$ *and* $\mathbb{P}_{\mathbb{C}}$ to be rewritten.

5.1 Role of type variables

Location of type variables We restrict type variables to appearing in type annotations for field access and method call, in field declarations, and as types of parameters to an instantiation expression (the $\bar{\mathtt{T}}$ in $\mathtt{new\ C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!)$). Originally, our intuition was that type variables would be introduced into bytecode during compilation of code that used classes unavailable to the compiler (as opposed to being mentioned explicitly in source code), so would only appear in field access and method call annotations. Gradually, we realised that there could be benefits in allowing type variables in the source code itself, *c.f.* to avoid naming implementation classes in section 2. As a limited example of this, we decided to allow type variables in field declarations; in future, they could also appear in method signatures (`X m(Y y) {...}`), as superclasses (`class A extends X {...}`), and within an instantiation expression ($\mathtt{new\ X}(...)(\!|...|\!)$).

Timing of type variable substitution Plainly, assumptions are the key mechanism that allow substitutions to happen later than verification - without assumptions, the verifier would have to substitute all type variables in order to check subtyping.

During execution, we require type variables in a field access/method call annotation to have been substituted before evaluation of the field access/method call. We do not require *all* the types of parameters in an instantiation's type annotation to be classes, because type variables in field declarations in a class `C` do not prevent the creation of a `C` object.[13]

When accessing a field, we require that if it was declared as a type variable, then that type variable has been substituted already. Other fields need not concern us, and can remain as type variables.

We could allow the execution of field access/method call containing *unsubstituted* type variables, provided type variables are allowed in method signatures. This is not reflected in the current system. Recently, we realised that we could even allow instantiation of an object whose type is a type variable: from $\mathtt{new\ X}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!)$ we obtain an object of type `X` whose fields are provided by $\bar{\mathtt{e}}$. This raises the question of what to assign the initial field values $\bar{\mathtt{e}}$ to, and further work is needed.

Scope of type variables The most general approach, taken in this work, is to have the scope of type variables be global. Thus, a type variable has the same meaning no matter which class it occurs in, *i.e.* it eventually substitutes to the same class. This is needed to allow a field declared as '`X f`' in one class to be accessed with $\mathtt{.f}(\!|\mathtt{C},\mathtt{X}|\!)$ from any other class. It also supports (if the syntax allows it) type variables in method signatures, *i.e.* `X m(Y y) {...}`, as `X`,`Y` in the invoking class are substituted the same as in the receiver.

Our original belief, before we considered fields declared as type variables, was that type variables would be introduced by the compiler and that the scope of type variables was naturally restricted to the method using them. Another possibility would be to restrict the scope of a type variable to a class and its superclasses, so that `X` in class `C` does not substitute to the same class as `X` in class $\mathtt{C}'$. We later realised that both approaches are unnecessarily restrictive and complicated.

5.2 Design of the formal system

Imperative v. functional model We originally thought that imperative features were needed to demonstrate that the formal system described issues of significance for practical OO languages encompassing inheritance. However, the imperative features required many more mechanisms (stack, heap, notion of well formed stack, heap and object) which in some sense detracted from the main issues. The restriction to a functional model shortened the operational semantics, *e.g.* with no heap, we can use the same typing rules for verification and run-time typing. This allowed us to concentrate on the central issues of type variables and substitution; we shall consider an imperative model later.

[13] The heap space required to allocate an object does not depend on the types of its fields.

We dropped FJ-style explicit constructors from the bytecode to hide the rather mechanical work of assigning values to fields at object instantiation. However, reflecting field assignment would allow us to precisely model the timing of substitution for type variables used as field types. This supports further work on an imperative model.

Substitutions In our current system we apply substitutions immediately to the whole program and expression. This is significantly simpler than our first approach, which was to carry substitutions around in the program.[14] This meant that we needed to consider whether to apply the substitutions in each judgment, and the distinction between the substitutions and their transitive closure became rather complex.

Note that substitutions are not remembered, so we may substitute `X` to `C` (hence no occurrences of `X` remain in the program) and then later, having loaded bytecode that mentions `X` again, substitute `X` to a different class $\texttt{C}'$.

Subtyping The decision of whether to use definite or assumptive subtypes was sometimes difficult. When checking the validity of class loading, we use an assumptive subtype about a class before it is loaded and a definite subtype about it after it is loaded, reflecting the fact that assumptions about a class must be true after it is loaded. For class verification (and the typing rules that it calls), it is obvious to use assumptive subtypes because we want to allow type variables to be substituted later than verification.

More interestingly, when extending a program by adding an assumption that involves a class, we use assumptive subtypes in order to be conservative about the class hierarchy. In the pre-condition to rule 3 of fig. 5, using definite subtypes would require that the supertype `T` and its superclasses have been loaded already; this is too eager. By using assumptive subtypes, we ensure that the class hierarchy *assumed* to exist so far (*i.e.* the hierarchy induced by assumptive subtypes) is not allowed to become cyclic. If some classes are already loaded, then definite subtype judgements about them can be used by the pre-condition, thanks to rule 5 in fig. 3.

6 Related and further work

Our inspiration for using type variables to support dynamic linking comes from discussions with Davide Ancona and Elena Zucca; from [SA93], which uses them to express requirements on classes during compilation; and from [AZ04], which suggests bytecode contain "type variables that must be instantiated during static inter-checking". The operational semantics presented here is based on [DLE03], and inherits the notion that program extension happens non-deterministically.

Recent work on the logical foundation for dynamic linking [AGW04] also considers the global substitution of type variables. In contrast to the high level at which we have worked, [Fra97] gives a clear exposition of the low-level issues involved in dynamic linking. Incorporating the different linking strategies it describes into our non-deterministic semantics would allow a wide range of execution environments to be modelled, including those for procedural languages.

The design of the system is delicate and is an ongoing process. Further work includes full proofs of soundness and progress for the functional system in this paper; flexible linking in an imperative model with heaps and assignment; compilation that emits bytecode with type variables; and implementation of a flexible dynamic linker in a modern virtual machine.

Acknowledgements We would like to thank Davide Ancona for suggesting the use of Featherweight Java to simplify the formal system, and John Knottenbelt, Christopher Anderson, Matthew Smith, Robert Chatley, William Heaven and Susan Eisenbach for helpful discussions.

[14] We also allowed substitutions that mapped type variables to other type variables. This allowed the system to automatically generate a substitution $\texttt{X} \mapsto \texttt{Y}$ if `X` was an assumptive subtype of `Y` and `Y` was an assumptive subtype of `X`.

References

[ADDZ04] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Even More Principal Typings for Java-like Languages. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 2004.

[AGW04] Martin Abadi, Georges Gonthier, and Benjamin Werner. Choice in Dynamic Linking. In *Proceedings of the 7th International Conference FOSSACS 2004 (ETAPS 2004)*, volume 2987 of *LNCS*, pages 12–26, Barcelona, Spain, March 2004. Springer-Verlag.

[AZ04] Davide Ancona and Elena Zucca. Principal Typings for Java-like Languages. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 2004)*, pages 306–317, Venice, Italy, 2004.

[DLE03] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

[Fra97] Michael Franz. Dynamic Linking of Software Components. *IEEE Computer*, pages 74–81, March 1997.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 132–146, Denver, Colorado, USA, 1999.

[JCF02] Sun Microsystems Inc. *The Java Collections Framework*, 2002. `http://java.sun.com/j2se/1.4.2/docs/guide/collections/`.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1999.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java Class Loading. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2000)*, pages 325–336, Minneapolis, Minnesota, USA, 2000.

[SA93] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'93)*, pages 439–450, Charleston, South Carolina, USA, 1993.

Appendix A - The Formal System

$$
\begin{array}{lll}
\mathbb{P} & ::= & (\mathbb{P}_{\mathbb{A}}, \mathbb{P}_{\mathbb{B}}, \mathbb{P}_{\mathbb{C}}) \\
\mathbb{P}_{\mathbb{A}} & ::= & (\mathtt{T}, \mathtt{T})^* \\
\mathbb{P}_{\mathbb{B}} & ::= & (\mathtt{C}, \mathtt{m}, \mathtt{T}, \mathtt{T}, \mathtt{e})^* \\
\mathbb{P}_{\mathbb{C}} & ::= & (\mathtt{C}, \mathtt{CL})^* \\
\mathtt{CL} & ::= & \texttt{class C extends C \{ } \bar{\mathtt{T}}\ \bar{\mathtt{f}}\texttt{; } \bar{\mathtt{M}}\ \texttt{\}} \\
\mathtt{M} & ::= & \texttt{C m(C y) \{ return e; \}} \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{X} \\
\mathtt{e} & ::= & \texttt{new C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!) \mid \texttt{.f}(|\mathtt{T}, \mathtt{T}|) \mid \texttt{.m}(|\mathtt{T}, \mathtt{T}, \mathtt{e}|) \mid \mathtt{y}
\end{array}
$$

Figure 2: Definitions

(1)
$$\frac{\mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \texttt{class C extends C}'\ \{\ldots\}}{\mathbb{P}\ \vdash\ \mathtt{C} \leq \mathtt{C}'}$$

(2)
$$\frac{}{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T}}$$

(3)
$$\frac{\mathbb{P}\ \vdash\ \mathtt{T}'' \leq \mathtt{T}' \quad \mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T}''}{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T}'}$$

(4)
$$\frac{(\mathtt{T}, \mathtt{T}') \in \mathbb{P}_{\mathbb{A}}}{\mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}'}$$

(5)
$$\frac{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T}'}{\mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}'}$$

(6)
$$\frac{\mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T}'' \leq \mathtt{T}' \quad \mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}''}{\mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}'}$$

Figure 3: Subtyping

(1)
$$\frac{}{\mathbb{P}, \Gamma\ \vdash\ \mathtt{y} : \Gamma(\mathtt{y})}$$

(2)
$$\frac{\mathbb{P}, \Gamma\ \vdash\ \bar{\mathtt{e}} : \bar{\mathtt{T}}'' \qquad \mathbb{P}\ \vdash_{\mathbb{A}}\ \bar{\mathtt{T}}'' \leq \bar{\mathtt{T}}'}{\mathbb{P}, \Gamma\ \vdash\ \texttt{new T}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}'|\!) : \mathtt{T}}$$

(3)
$$\frac{\mathbb{P}, \Gamma\ \vdash\ \mathtt{e} : \mathtt{T} \qquad \mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}_d}{\mathbb{P}, \Gamma\ \vdash\ \mathtt{e.f}(|\mathtt{T}_d, \mathtt{T}_f|) : \mathtt{T}_f}$$

(4)
$$\frac{\mathbb{P}, \Gamma\ \vdash\ \mathtt{e} : \mathtt{T} \quad \mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T} \leq \mathtt{T}_d \quad \mathbb{P}, \Gamma\ \vdash\ \mathtt{e}' : \mathtt{T}' \quad \mathbb{P}\ \vdash_{\mathbb{A}}\ \mathtt{T}' \leq \mathtt{T}_p}{\mathbb{P}, \Gamma\ \vdash\ \mathtt{e.m}(|\mathtt{T}_d, \mathtt{T}_r, \mathtt{T}_p|)(\mathtt{e}') : \mathtt{T}_r}$$

Figure 4: Typing and Verification

$$(\mathbb{P}_\mathbb{A}, \mathbb{P}_\mathbb{B}, \mathbb{P}_\mathbb{C}) \oplus \mathtt{CL_C} = (\mathbb{P}_\mathbb{A}, \mathbb{P}_\mathbb{B}, (\mathbb{P}_\mathbb{C}, (\mathtt{C}, \mathtt{CL_C})))$$
$$(\mathbb{P}_\mathbb{A}, \mathbb{P}_\mathbb{B}, \mathbb{P}_\mathbb{C}) \oplus \mathtt{CV_C} = (\mathbb{P}_\mathbb{A}, (\mathbb{P}_\mathbb{B}, (\mathtt{C}, \mathtt{CV_C})), \mathbb{P}_\mathbb{C})$$
$$(\mathbb{P}_\mathbb{A}, \mathbb{P}_\mathbb{B}, \mathbb{P}_\mathbb{C}) \oplus (\mathtt{T}, \mathtt{T}') = ((\mathbb{P}_\mathbb{A}, (\mathtt{T}, \mathtt{T}')), \mathbb{P}_\mathbb{B}, \mathbb{P}_\mathbb{C})$$

(1)
$$\frac{\begin{array}{l}\mathtt{CL_C} = \mathtt{class\ C\ extends\ C'\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}} \\ \mathbb{P}_\mathbb{C}(\mathtt{C}) = \epsilon, \quad \mathbb{P}_\mathbb{C}(\mathtt{C}') \neq \epsilon \\ \mathbb{P} \vdash_\mathbb{A} \mathtt{C} \leq \mathtt{T} \implies \mathbb{P} \oplus \mathtt{CL_C} \vdash \mathtt{C} \leq \mathtt{T}\end{array}}{\vdash \mathbb{P} \oplus \mathtt{CL_C} \leq \mathbb{P}}$$

(2)
$$\frac{\begin{array}{l}\mathbb{P}_\mathbb{C}(\mathtt{C}) = \mathtt{class\ C\ extends\ C'\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}} \\ \forall i : 1..n \quad \mathtt{M}^i = {\mathtt{C}_r}^i\ \mathtt{m}^i({\mathtt{C}_p}^i\ \mathtt{y}^i)\ \{\ \mathtt{return\ e}^i;\ \} \implies \\ \qquad \mathbb{P}, \{\mathtt{this} : \mathtt{C}, \mathtt{y}^i : {\mathtt{C}_p}^i\} \vdash \mathtt{e}^i : \mathtt{T}^i \ \wedge \ \mathbb{P} \vdash_\mathbb{A} \mathtt{T}^i \leq {\mathtt{C}_r}^i \\ \mathbb{P}_\mathbb{B}(\mathtt{C}) = \epsilon \\ \mathtt{CV_C} = \mathbb{P}_\mathbb{B}(\mathtt{C}') \bullet (\mathtt{m}^i, {\mathtt{C}_r}^i, {\mathtt{C}_p}^i \mapsto \mathtt{e}^i)^{i=1..n}\end{array}}{\vdash \mathbb{P} \oplus \mathtt{CV_C} \leq \mathbb{P}}$$

(3)
$$\frac{\mathbb{P} \vdash_\mathbb{A} \mathtt{T} \leq \mathtt{C}' \implies \mathbb{P} \not\vdash_\mathbb{A} \mathtt{C}' \leq \mathtt{C}}{\vdash \mathbb{P} \oplus (\mathtt{C}, \mathtt{T}) \leq \mathbb{P}}$$

(4)
$$\frac{}{\vdash \mathbb{P} \oplus (\mathtt{X}, \mathtt{T}) \leq \mathbb{P}}$$

(5)
$$\frac{\begin{array}{l}\vdash \mathbb{P}'' \leq \mathbb{P} \\ \vdash \mathbb{P}' \leq \mathbb{P}''\end{array}}{\vdash \mathbb{P}' \leq \mathbb{P}}$$

(6)
$$\frac{}{\vdash \mathbb{P} \leq \mathbb{P}}$$

Figure 5: Program extension

$$\mathit{fields}(\mathbb{P}, \mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \iff \mathbb{P}_\mathbb{C}(\mathtt{C}) = \mathtt{class\ C} \ldots \{\ \bar{\mathtt{T}}\ \bar{\mathtt{f}}; \ldots\}$$

(1)
$$\frac{\begin{array}{l}\mathit{fields}(\mathbb{P}, \mathtt{C}_d) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \\ \mathtt{T}_i = \mathtt{C}_f \quad \mathtt{f}_i = \mathtt{f} \quad (\mathtt{T}_j = \mathtt{C}_f \ \wedge \ \mathtt{f}_j = \mathtt{f} \implies j \leq i)\end{array}}{\mathbb{P}, \mathtt{new\ C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!).\mathtt{f}(\!|\mathtt{C}_d, \mathtt{C}_f|\!) \rightsquigarrow \mathbb{P}, \mathtt{e}_i}$$

(2)
$$\frac{\mathbb{P}_\mathbb{B}(\mathtt{C})(\mathtt{m}, \mathtt{C}_r, \mathtt{C}_p) = \mathtt{e}}{\mathbb{P}, \mathtt{new\ C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!).\mathtt{m}(\!|\mathtt{C}_d, \mathtt{C}_r, \mathtt{C}_p|\!)(\mathtt{d}) \rightsquigarrow \mathbb{P}, \mathtt{e}[\mathtt{new\ C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!)/\mathtt{this}][\mathtt{d}/\mathtt{y}]}$$

(3)
$$\frac{\vdash \mathbb{P}' \leq \mathbb{P}}{\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}', \mathtt{e}}$$

(4)
$$\frac{\mathbb{P} \vdash \sigma \diamond}{\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}[\sigma], \mathtt{e}[\sigma]}$$

Figure 6: Execution

$$\frac{\begin{array}{l}\text{(1)}\\ \quad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{X} \implies \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{C} \\ \quad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{X} \leq \mathtt{T} \implies \mathbb{P} \vdash_{\mathbb{A}} \mathtt{C} \leq \mathtt{T}\end{array}}{\mathbb{P} \vdash \mathtt{C/X} \diamond} \qquad \frac{\text{(2)}}{\mathbb{P} \vdash \mathtt{T/T} \diamond}$$

Figure 7: Well-formed substitutions

$$\frac{\begin{array}{l}\text{(1)}\\ \quad \mathtt{Object} \notin dom(\mathbb{P}_{\mathbb{C}}) \\ \quad \mathbb{P} \vdash \mathtt{C} \leq \mathtt{C}' \implies \mathtt{C}' \in dom(\mathbb{P}_{\mathbb{C}}) \\ \quad \mathbb{P} \vdash \mathtt{C} \leq \mathtt{C}' \ \wedge \ \mathbb{P} \vdash \mathtt{C}' \leq \mathtt{C} \implies \mathtt{C} = \mathtt{C}' \\ \quad \forall \mathtt{C} \in dom(\mathbb{P}_{\mathbb{C}}) : \\ \qquad (\mathbb{P}_{\mathbb{B}}(\mathtt{C}) = \epsilon) \ \vee \\ \qquad (\mathbb{P}_{\mathbb{B}}(\mathtt{C})(\mathtt{m}, \mathtt{T}_r, \mathtt{T}_p) = \mathtt{e} \implies \mathbb{P}, \{\mathtt{this} : \mathtt{C}, \mathtt{y} : \mathtt{T}_p\} \vdash \mathtt{e} : \mathtt{T}' \ \wedge \ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T}' \leq \mathtt{T}_r)\end{array}}{\vdash \mathbb{P}}$$

Figure 8: Well-formed programs

Appendix B - Properties of the Formal System

Conjecture 1 Execution causes program extension modulo substitution.

$\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}', \mathtt{e}' \implies \exists \sigma\text{: } \mathbb{P} \vdash \sigma \diamond$ and $\vdash \mathbb{P}' \leq \mathbb{P}[\sigma]$

Conjecture 2 Well-formed substitutions preserve types and program well-formedness.

If $\mathbb{P} \vdash \sigma \diamond$ then

- $\mathbb{P} \vdash \mathtt{T}' \leq \mathtt{T} \implies \mathbb{P}[\sigma] \vdash \mathtt{T}'[\sigma] \leq \mathtt{T}[\sigma]$
- $\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T} \implies \mathbb{P}[\sigma], \Gamma \vdash \mathtt{e}[\sigma] : \mathtt{T}[\sigma]$
- $\vdash \mathbb{P} \implies \vdash \mathbb{P}[\sigma]$
- $\mathbb{P}[\sigma] \vdash \sigma \diamond$

Conjecture 3 Program extension preserves types and program well-formedness.

If $\vdash \mathbb{P}' \leq \mathbb{P}$ then

- $\mathbb{P} \vdash \mathtt{T}' \leq \mathtt{T} \implies \mathbb{P}' \vdash \mathtt{T}' \leq \mathtt{T}$
- $\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T} \implies \mathbb{P}', \Gamma \vdash \mathtt{e} : \mathtt{T}$
- $\vdash \mathbb{P} \implies \vdash \mathbb{P}'$
- $\mathbb{P} \vdash \sigma \diamond \implies \mathbb{P}' \vdash \sigma \diamond$

Conjecture 4 Execution preserves types of unevaluated expressions.

If $\vdash \mathbb{P}$ and $\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T}$ and $\mathbb{P}, \mathtt{e} \rightsquigarrow \mathbb{P}', \mathtt{e}'$

then

$\vdash \mathbb{P}'$ and

$\mathbb{P}, \Gamma \vdash \mathtt{e}'' : \mathtt{T}'' \implies \exists \sigma\text{: } \mathbb{P} \vdash \sigma \diamond$ and $\mathbb{P}', \Gamma \vdash \mathtt{e}''[\sigma] : \mathtt{T}''[\sigma]$